STRYKERMANAGEMENTAPP

Code analysis

By: Gateway

2022-03-29

CONTENT

INTRODUCTION

This document contains results of the code analysis of StrykerManagementApp.

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java]; Stryker [Kotlin]; Stryker [XML];
  - Files: AX9JLBXVZWz9exVyjg9p.json; AX_WRsJAZoNvhpLrXfw9.json; AX_WbTE_ZoNvhpLrXf1b.json;
- Quality Gate
  - Name: Sonar-StrykerAndroid
  - File: Sonar-StrykerAndroid.xml

StrykerManagementApp

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
A	A	E	A

QUALITY GATE STATUS

Quality Gate Status	Failed

Metric	Value
Reliability Rating on New Code	OK
Security Rating on New Code	OK
Maintainability Rating on New Code	OK
Duplicated Lines (%) on New Code	OK
Security Hotspots Reviewed on New Code	ERROR (0.0% is less than 100%)

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
0.0 %	4.6 %	8.5 %	44.0	99.3 %

TESTS

Total	Success Rate	Skipped	Errors	Failures

0	**0 %**	**0**	**0**	**0**

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
-	-	1d 6h 37min	**1d 6h 37min**

METRICS RANGE

	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	1.0	0.0	0.0	0.0
Max	1197.0	1141.0	12921.0	96.9	0.0	91.6

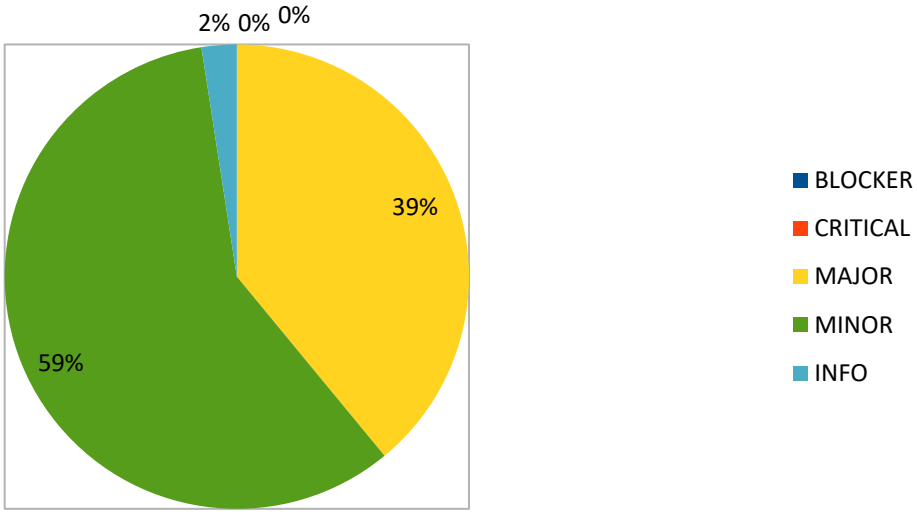
VOLUME

Language	Number
Java	837
Kotlin	10660
XML	1424
Total	12921

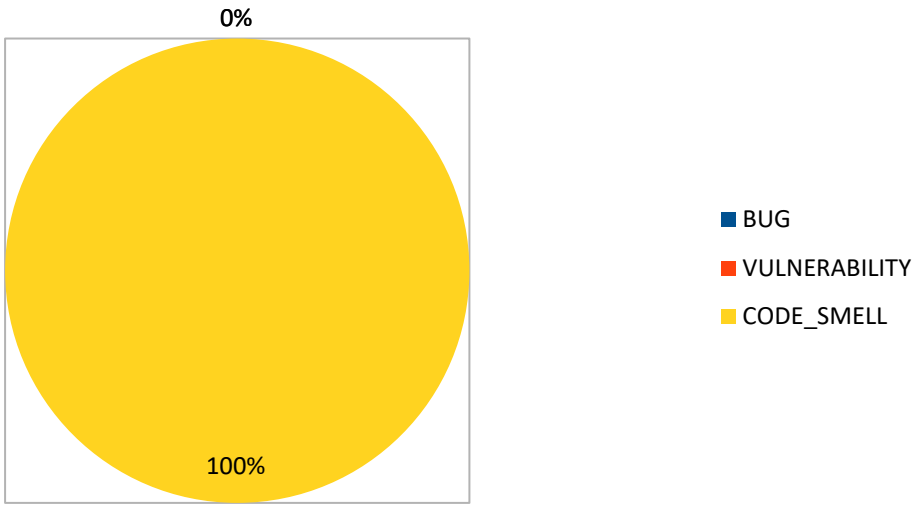
ISSUES

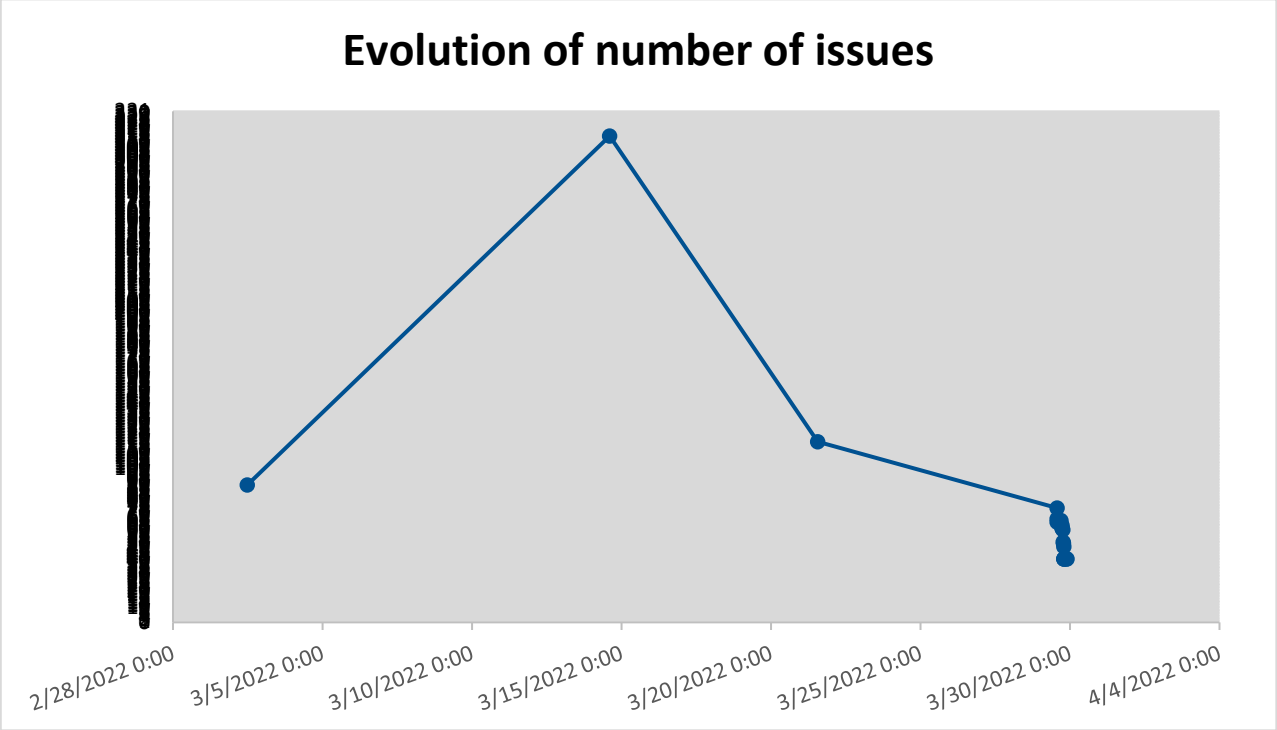
CHARTS

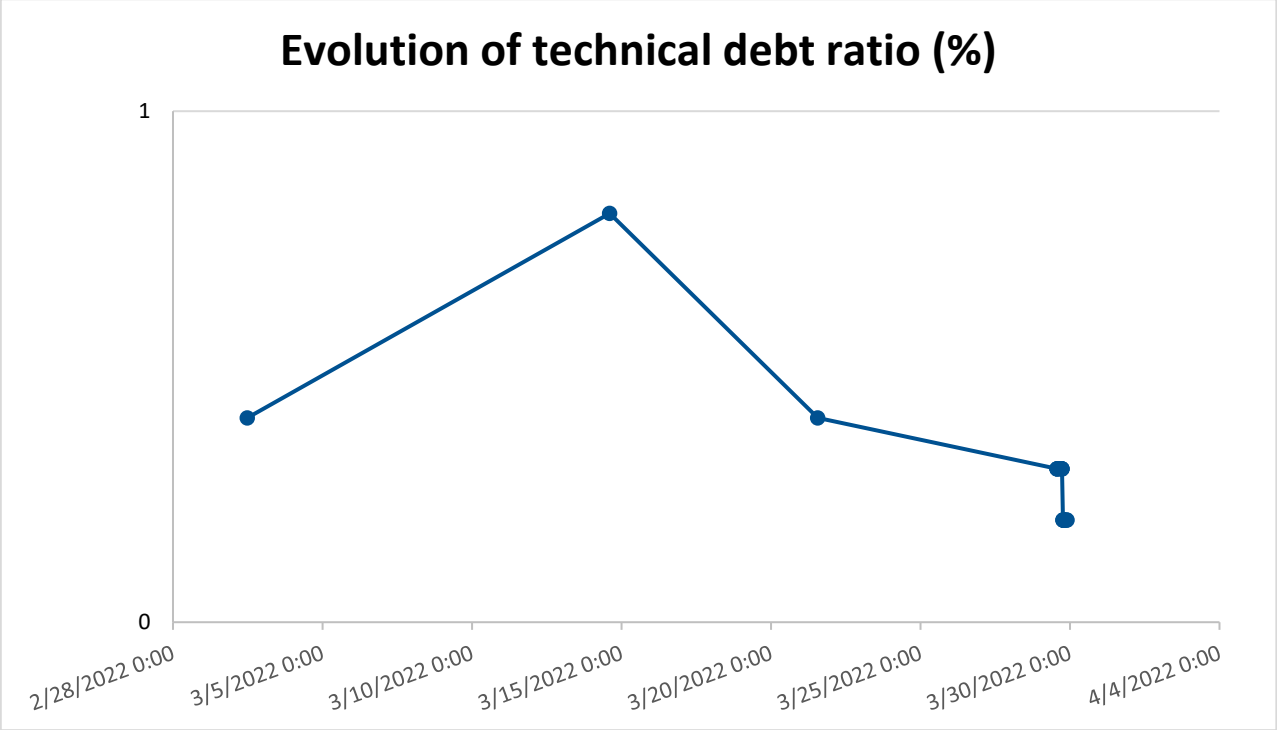
Number of issues by severity



Number of issues by type




Evolution of number of issues
2/28/2022 0:00
3/5/2022 0:00
3/10/2022 0:00
3/15/2022 0:00
3/20/2022 0:00
3/25/2022 0:00
3/30/2022 0:00
4/4/2022 0:00


Evolution of technical debt ratio (%)
1
0
2/28/2022 0:00
3/5/2022 0:00
3/10/2022 0:00
3/15/2022 0:00
3/20/2022 0:00
3/25/2022 0:00
3/30/2022 0:00
4/4/2022 0:00

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	0	0	0	0
VULNERABILITY	0	0	0	0	0
CODE_SMELL	1	24	16	0	0

ISSUES LIST

Name	Description	Type	Severity	Number
Track uses of "TODO" tags	TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example fun doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment	CODE_SMELL	INFO	1
Functions should not have too many parameters	A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things. Noncompliant Code Example With a maximum number of 4 parameters: fun foo(p1: String, p2: String, p3: String, p4: String, p5: String) { // Noncompliant // ... } Compliant Solution fun foo(p1: String, p2: String, p3: String, p4: String) { // ... } Exceptions Methods annotated with Spring's @RequestMapping (and related shortcut annotations, like @GetRequest) or @JsonCreator may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored.	CODE_SMELL	MAJOR	2
Coroutine usage should adhere to structured concurrency principles	Kotlin coroutines follow the principle of structured concurrency. This helps in preventing resource leaks and ensures that scopes are only exited once all child coroutines have exited. Hence, structured concurrency enables developers to build concurrent applications while having to worry less about cleaning up concurrent tasks manually. It is possible to break this concept of structured concurrency in various ways. Generally, this is not advised, as it can open the door to coroutines being leaked or lost. Ask yourself if breaking structured concurrency here is really necessary for the application's business logic, or if it could be avoided by refactoring parts of the code. This	CODE_SMELL	MAJOR	7

	rule raises an issue when it detects that the structured concurrency principles are violated. It avoids reporting on valid use cases and in situations where developers have consciously opted into using delicate APIs (e.g. by using the @OptIn annotation) and hence should be aware of the possible pitfalls. Noncompliant Code Example GlobalScope: fun main() { GlobalScope.launch { // Noncompliant: no explicit opt-in to DelicateCoroutinesApi // Do some work }.join() } Manual job instantiation: fun startLongRunningBackgroundJob(job: Job) { val coroutineScope = CoroutineScope(job) coroutineScope.launch(Job()) { // Noncompliant: new job instance passed to launch() // Do some work } } Manual supervisor instantiation: coroutineScope { launch(SupervisorJob()) { // Noncompliant: new supervisor instance passed to launch() // Do some work } } Compliant Solution In many situations, a good pattern is to use coroutineScope as provided in suspending functions: suspend fun main() { worker() } suspend fun worker() { coroutineScope { launch { // Compliant: no manually created job/supervisor instance passed to launch() // Do some work } } } GlobalScope: @OptIn(DelicateCoroutinesApi::class) fun main() { GlobalScope.launch { // Compliant: explicit opt-in to DelicateCoroutinesApi via method annotation // Do some work }.join() } No manual job instantiation: fun startLongRunningBackgroundJob(job: Job) { val coroutineScope = CoroutineScope(job) coroutineScope.launch { // Compliant: no manually created job/supervisor instance passed to launch() // Do some work } } Using a supervisor scope instead of manually instantiating a supervisor: supervisorScope { launch { // Do some work } } See Structured concurrency in the Kotlin docs GlobalScope documentation coroutineScope documentation Android coroutines best practices		
Dispatchers should be injectable	Dispatchers should not be hardcoded when using withContext or creating new coroutines using launch or async. Injectable dispatchers ease testing by allowing tests to inject more deterministic dispatchers. You can use default values for the dispatcher constructor arguments to eliminate the need to specify them explicitly in the production caller contexts. This rule raises an issue when it finds a hard-coded dispatcher being used in withContext or when starting new coroutines. Noncompliant Code Example class ExampleClass { suspend fun doSomething() { withContext(Dispatchers.Default) { // Noncompliant: hard-coded dispatcher ... } } } Compliant Solution class ExampleClass(private val	CODE_SMELL	MAJOR	7

	dispatcher: CoroutineDispatcher = Dispatchers.Default) { suspend fun doSomething() { withContext(dispatcher) { ... } } } See Inject dispatchers (Android coroutines best practices)			
Unnecessary imports should be removed	The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Unused and useless imports should not occur if that is the case. Leaving them in reduces the code's readability, since their presence can be confusing. Noncompliant Code Example package my.company; import java.lang.String; // Noncompliant; java.lang classes are always implicitly imported import my.company.SomeClass; // Noncompliant; same-package files are always implicitly imported import java.io.File; // Noncompliant; File is not used import my.company2.SomeType; import my.company2.SomeType; // Noncompliant; 'SomeType' is already imported class ExampleClass { public String someString; public SomeType something; } Exceptions Imports for types mentioned in Javadocs are ignored.	CODE_SMELL	MINOR	1
"@Deprecated" code should not be used	Once deprecated, classes, and interfaces, and their members should be avoided, rather than used, inherited or extended. Deprecation is a warning that the class or interface has been superseded, and will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology. Noncompliant Code Example /** * @deprecated As of release 1.3, replaced by {@link #Fee} */ @Deprecated public class Fum { ... } public class Foo { /** * @deprecated As of release 1.7, replaced by {@link #doTheThingBetter()} */ @Deprecated public void doTheThing() { ... } public void doTheThingBetter() { ... } } public class Bar extends Foo { public void doTheThing() { ... } // Noncompliant; don't override a deprecated method or explicitly mark it as @Deprecated } public class Bar extends Fum { // Noncompliant; Fum is deprecated public void myMethod() { Foo foo = new Foo(); // okay; the class isn't deprecated foo.doTheThing(); // Noncompliant; doTheThing method is deprecated } } See MITRE, CWE-477 - Use of Obsolete Functions CERT, MET02-J. - Do not use deprecated or obsolete classes or methods	CODE_SMELL	MINOR	3
Code annotated as deprecated should not be used	Code annotated as deprecated should not be used since it will be removed sooner or later. Noncompliant Code Example @Deprecated("") interface Old class Example :	CODE_SMELL	MINOR	20

```
Old // Noncompliant
```

SECURITY HOTSPOTS

SECURITY HOTSPOTS COUNT BY CATEGORY AND PRIORITY

Category / Priority	LOW	MEDIUM	HIGH
LDAP Injection	0	0	0
Object Injection	0	0	0
Server-Side Request Forgery (SSRF)	0	0	0
XML External Entity (XXE)	0	0	0
Insecure Configuration	1	0	0
XPath Injection	0	0	0
Authentication	0	0	0
Weak Cryptography	0	0	0
Denial of Service (DoS)	0	0	0
Log Injection	0	0	0
Cross-Site Request Forgery (CSRF)	0	0	0
Open Redirect	0	0	0
SQL Injection	0	0	0
Buffer Overflow	0	0	0
File Manipulation	0	0	0
Code Injection (RCE)	0	0	0
Cross-Site Scripting (XSS)	0	0	0
Command Injection	0	0	0
Path Traversal Injection	0	0	0

HTTP Response Splitting	0	0	0
Others	6	0	0

SECURITY HOTSPOTS LIST

Category	Name	Priority	Severity	Count
Insecure Configuration	Delivering code in production with debug features activated is security-sensitive	LOW	MINOR	1
	Using clear-text protocols is security-sensitive	LOW	CRITICAL	1
Others	Receiving intents is security-sensitive	LOW	CRITICAL	2
Others	Allowing application backup is security-sensitive	LOW	MAJOR	1
Others	Accessing Android external storage is security-sensitive	LOW	CRITICAL	3